May 1995

UILU-ENG-95-2211
CRHC-95-10

Center for Reliable and High Performance Computing

Recoverable Distributed Shared Memory Under Sequential and Relaxed Consistency

Bob Janssens and W. Kent Fuchs

Coordinated Science Laboratory
College of Engineering
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois 6b. OFFICE SYMBOL (*If applicable*) N/A	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research and NASA
6c. ADDRESS (*City, State, and ZIP Code*) 1308 W. Main ST. Urbana, IL 61801	7b. ADDRESS (*City, State, and ZIP Code*) Ames Research Ctr. Moffett Field, CA 800 N. Quincy St. Arlington, VA 22217
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Joint Services Electronics Program 7a. 8b. OFFICE SYMBOL (*If applicable*)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00014- 90-J-1270 NASA NAG 1-613
8c. ADDRESS (*City, State, and ZIP Code*) 800 N. Quincy St. Arlington, VA 22217 7b.	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (*Include Security Classification*)
Recoverable Distributed Shared Memory Under Sequential and Relaxed Consistency

12. PERSONAL AUTHOR(S)
Bob Janssens and W. Kent Fuchs

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (*Year, Month, Day*)	15. PAGE COUNT
Technical	FROM _____ TO _____	May 1995	29

16. SUPPLEMENTARY NOTATION

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (*Continue on reverse if necessary and identify by block number*)
checkpointing and rollback recovery, distributed shared memory, memory consistency models

19. ABSTRACT (*Continue on reverse if necessary and identify by block number*)

Distributed shared memory (DSM) implemented on a cluster of workstations is an increasingly attractive platform for executing parallel scientific applications. Checkpointing and rollback techniques can be used in such a system to allow the computation to progress in spite of the temporary failure of one or more processing nodes. The complexity and overhead inherent in traditional message-passing checkpointing techniques can be reduced by taking advantages of specific properties of DSM. In this paper we show that, if designed correctly, a DSM system only needs to consider a subset of message-passing dependencies for correct rollback. A passive server model of DSM computation is described that allows a loosening of dependency restrictions by considering the events involved in interactions between nodes as atomic. An ownership timestamp scheme is used to eliminate many of the dependencies related to keeping directories consistent. The schemes can be implemented in DSM hardware by simply redesigning the directory at the network interface. Finally, we show that by relaxing the memory consistency model and using lazy release consistency, it is possible to further relax dependency restrictions.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (*Include Area Code*) / 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

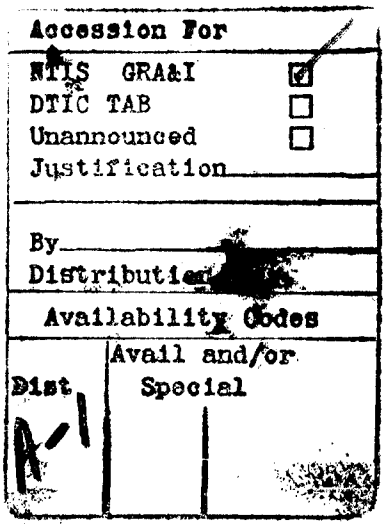
SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

RECOVERABLE DISTRIBUTED SHARED MEMORY UNDER SEQUENTIAL AND RELAXED CONSISTENCY

Bob Janssens and W. Kent Fuchs

Center for Reliable and High-Performance Computing
Coordinated Science Laboratory
University of Illinois
1308 West Main Street
Urbana, IL 61801

April, 1995



Abstract

Distributed shared memory (DSM) implemented on a cluster of workstations is an increasingly attractive platform for executing parallel scientific applications. Checkpointing and rollback techniques can be used in such a system to allow the computation to progress in spite of the temporary failure of one or more processing nodes. The complexity and overhead inherent in traditional message-passing checkpointing techniques can be reduced by taking advantages of specific properties of DSM. In this paper we show that, if designed correctly, a DSM system only needs to consider a subset of message-passing dependencies for correct rollback. A passive server model of DSM computation is described that allows a loosening of dependency restrictions by considering the events involved in interactions between nodes as atomic. An ownership timestamp scheme is used to eliminate many of the dependencies related to keeping directories consistent. The schemes can be implemented in DSM hardware by simply redesigning the directory at the network interface. Finally, we show that by relaxing the memory consistency model and using lazy release consistency, it is possible to further relax dependency restrictions.

Keywords: checkpointing and rollback recovery, distributed shared memory, memory consistency models.

This paper contains material previously presented in in the 23rd FTCS [24] and the 13th SRDS [25]. This research was supported in part by the Office of Naval Research under contract N00014-91-J-1283, and by the National Aeronautics and Space Administration (NASA) under Grant NASA NAG 1-613, in cooperation with the Illinois Computer Laboratory for Aerospace Systems and Software (ICLASS).

1 Introduction

Increasingly, massively parallel supercomputers employ off-the-shelf workstations linked by a high-speed network as building blocks [10, 38]. Distributed shared memory (DSM) provides the advantages of a shared memory image in such systems. An all-software implementation of DSM has been shown to approach the performance of a bus-based multiprocessor for a small number of processors [11]. For more scalable performance, special purpose directory and interconnect hardware can be used to implement DSM [29, 35].

Checkpointing and rollback are commonly used to recover from detected processor errors in environments where high reliability is essential. The use of workstation clusters for parallel scientific computing makes it desirable to have the ability to roll back to a saved state even when reliability demands are less critical. In a network, a workstation may kill a process or completely reboot, either due to a system exception, or due to direct action by a user. In a networked workstation environment with multiple users, long-running applications should not adversely impact other users. One method that can be used to ensure this is checkpointing and recovery, as implemented in the Condor environment [32]. It is therefore desirable that long-running parallel applications be able to roll back the computation on a processing node by restarting from a checkpoint. In parallel systems, dependencies between processing nodes can cause the overall system state to be incorrect when one node rolls back. The problem of rolling back to a *consistent global state* has been widely investigated for message-passing systems. It is possible to directly apply this research to shared memory, by modeling the system in terms of message passing. However, most previous work in shared-memory recovery has used a laxer model of dependencies, using the intuition that only messages that transfer actual application data should cause dependencies. This assumption simplifies the implementation of a recoverable DSM, since many control messages do not need to be tracked. Furthermore, the performance overhead of handling dependencies is reduced, and the potential for rollback propagation is decreased.

In this paper, we develop a design framework for recoverable DSM which guarantees that only data transfer dependencies need to be considered. We use a a *passive server model* of DSM, where interaction events between processes are considered atomic. Together with an *ownership timestamp* scheme which recovers directory information independently, this model allows restriction of dependencies to the actual transfer

of blocks of data between nodes. Additionally, these dependencies are unidirectional, eliminating the need to log in-transit messages. We also develop a scheme that uses a relaxed memory consistency model to further limit the pattern of dependencies to only synchronization interactions. To avoid introducing spurious dependencies due to false sharing, it is necessary to use lazy release consistency with multiple writers. The complexity of this scheme makes it applicable to software DSM but unsuitable for hardware implementation.

Once the dependency pattern has been defined, any of the well-known techniques for distributed rollback recovery can be applied to DSM. In this paper, we therefore focus on the basic problem of determining exactly which dependencies need to be considered in any rollback recovery scheme for DSM.

2 Rollback to a Consistent State in Parallel Systems

To enable rollback recovery in a parallel system, the state of each node is periodically saved as a checkpoint. When one or more nodes roll back to a checkpoint, the resulting global state needs to be consistent, that is, it needs to be a state that could have existed if no rollback had occurred. The global state of a message-passing system contains any messages in transit, in addition to the the local state on each node [8]. In general non-deterministic execution, to maintain a consistent state, when a node rolls back past the sending of a message, the receiver must roll back to undo the effect of the message. Every message is therefore said to introduce a *dependency* from the sender to the receiver. When a node rolls back past the receiving of a message, and the sender does not roll back and re-send the message, some mechanism must be provided to retrieve the contents of the message. If such a logging mechanism is not available, the sender must also roll back.

Various methods exist to ensure recovery to a consistent state, differing in the way they handle dependencies. In globally *coordinated checkpointing* [14, 27, 30], when a node decides to checkpoint, it attempts to create a global consistent checkpoint by sending messages to all other nodes, telling them to take a tentative checkpoint. Because of propagation delay of the checkpointing messages it is possible that other messages sent at about the same time introduce dependencies, causing the tentative checkpoints to be inconsistent and therefore requiring adjustment in their locations. To handle the possibility of rollback during a checkpointing

session, it is necessary to implement a two-phase protocol, where tentative checkpoints are committed only when it is assured that every node has taken a permanent checkpoint. Since the resulting global checkpoint is consistent, correctness is assured by rolling back every node to the last checkpoint whenever one node needs to roll back. The coordination algorithm can be optimized by adding continuous dependency tracking. Then only processing nodes that have dependencies in the current checkpoint interval need to participate in checkpointing and rollback [27].

In *independent checkpointing*, no effort is made to coordinate the checkpointing on the different processing nodes [5, 41]. Therefore it is necessary to keep track of the pattern of dependencies between nodes to determine which need to follow suit when one node rolls back. In order to handle in-transit messages, it is necessary to log all communication in addition to tracking their dependencies. In this scheme, every dependency carrying message introduces a chance of rollback propagation, which can escalate into a domino effect. With loosely synchronized clocks, it is possible to use a *timestamp-based checkpointing* technique, where the taking of checkpoints at approximately the same time on every node limits the necessity of dependency tracking and message logging to a short time period around the time of checkpointing [12]. Some schemes use *communication-induced checkpointing*, where every dependency induces a checkpoint on other nodes, to eliminate rollback propagation.

Another class of recovery algorithms uses *logging and deterministic replay* to independently recover failed processing nodes [37]. These algorithms are restricted to systems where the computation is guaranteed to be piecewise deterministic. Since all interactions with other nodes are logged, and the execution is guaranteed to proceed exactly the same after rollback as before, no rollback propagation occurs if messages are logged synchronously. If an optimistic scheme is used, where messages are logged asynchronously, dependency tracking has to be used to guarantee correct recovery.

Various distributed system recovery techniques have been applied to shared memory. Communication-induced checkpointing is used in the Sequoia system [4], by Wu *et al.* in both bus-based multiprocessors [43] and software DSM systems [44], and by Janssens and Fuchs in DSM systems with relaxed consistency [24]. Ahmed *et al.* presented three schemes for bus-based systems that use globally coordinated, partially coordinated and communication-induced checkpointing respectively [2]. Banâtre *et al.* have proposed a scheme

that uses dependency tracking at the shared memory in a bus-based system to implement coordinated checkpointing [3]. In an extension of this scheme to a cache-only memory architecture (COMA) DSM, Gefflaut *et al.* use globally coordinated checkpointing [18]. Janakiraman and Tamir also presented a coordinated checkpointing scheme for DSM, eliminating some dependencies by using a *dirty-since-checkpoint* bit for every page [23]. Various DSM schemes based on logging and deterministic replay have also been proposed and implemented [15, 40, 34, 36].

It is intuitively obvious that the dependencies of message passing are too strict for shared-memory parallel programs. For instance, two reads by different processors to a shared variable with no intervening writes do not depend on each other even though both processors exchange messages with the shared memory element. In the literature on replay for debugging in shared memory systems, a dependency from memory access a to memory access b is generally said to exist if a accesses a shared variable that b later accesses, and at least one of the two accesses is a write [33]. Various papers have recently argued for treating a write as a two-way dependency with a memory element, while treating a read as a one-way dependency from the memory to the process [20, 22]. Therefore, there is no dependency from a read to a write if the read precedes the write.

This more relaxed dependency model can be used for rollback recovery only if there is no possibility of deadlock due to processes waiting for messages that may never arrive. In bus-based systems, where bounded transmission delay eliminates the need for acknowledgements, deadlock is avoided. In DSM systems, however, other measures have to be taken to avoid messaging deadlock. In the bus-based recovery scheme of Banâtre *et al,* a dependency is recorded between any processor that writes a data item and another that reads it. A bidirectional dependency is recorded between two processors that write a data item consecutively [17]. Wu *et al.*'s recovery scheme takes a checkpoint of the originating process *and* of the data item accessed on every data transfer between processing nodes [43, 44]. Janakiraman's scheme also considers only data transfers in determining dependencies, using an optimization where a transfer of data that has not been modified since the checkpoint does not cause a dependency [23]. The effect of all three schemes is to conform to the more relaxed dependence relation.

3 The Passive Server Model of DSM

In the message-passing model, the dependencies caused by every message dictate the design of methods that ensure rollback to a consistent state. Since intuitively it is clear that not all message-passing dependencies need to be considered in DSM, a new model is necessary to reason about consistent rollback in DSM.

Program execution in a message-passing distributed system is modeled as a set of processes and a set of reliable channels which the processes use to communicate with each other. Overall program execution is represented by a pair, $P = (E, \xrightarrow{D})$, where E is a set of events and $\xrightarrow{D}$ is the *dependence* relation defined over E. Events within a process are ordered by the $\xrightarrow{XO}$ (execution order) relation. Events on different processes are ordered by the $\xrightarrow{M}$ (message) relation where $a \xrightarrow{M} b$ means event a sent a message and event b received it. The $\xrightarrow{D}$ relation is the union of the other two: $\xrightarrow{D} = \xrightarrow{XO} \cup \xrightarrow{M}$. Every event represents an atomic action which may change the state in one of the processes. A special checkpoint event can be inserted between two events to record the current state of the process.

When a process needs to roll back, it notifies all other processes and rolls back to a checkpoint. Upon receiving notification of a rollback, a process can either roll back to a checkpoint, or continue operation. If it continues, we can treat the current volatile state as a virtual checkpoint [42]. A global checkpoint is a set of real and virtual checkpoints, one per process. Consider two events a and b, where b occurs in the execution order before the global checkpoint and a occurs in the execution order after the global checkpoint. A global checkpoint is consistent if there are no two such events such that $a \xrightarrow{M} b$ or $b \xrightarrow{M} a$. A global checkpoint is also consistent if lost messages can be retrieved during re-execution and there are no two events such that $a \xrightarrow{M} b$.

To simplify reasoning about consistency of global checkpoints it is useful to treat the $\xrightarrow{M}$ relation as bidirectional. To do this we replace every dependency $a \xrightarrow{M} b$, by a causal dependency $a \xrightarrow{C} b$, and a backward dependency $b \xrightarrow{B} a$. Consider again two events a and b, where b occurs in the execution order before a global checkpoint and a occurs in the execution order after a global checkpoint. The requirements for consistency are now that there are no two such events such that $a \xrightarrow{C} b$ and there are no two such events such that $b \xrightarrow{B} a$ and the message between a and b is unlogged. Figure 1 presents some examples of consistent

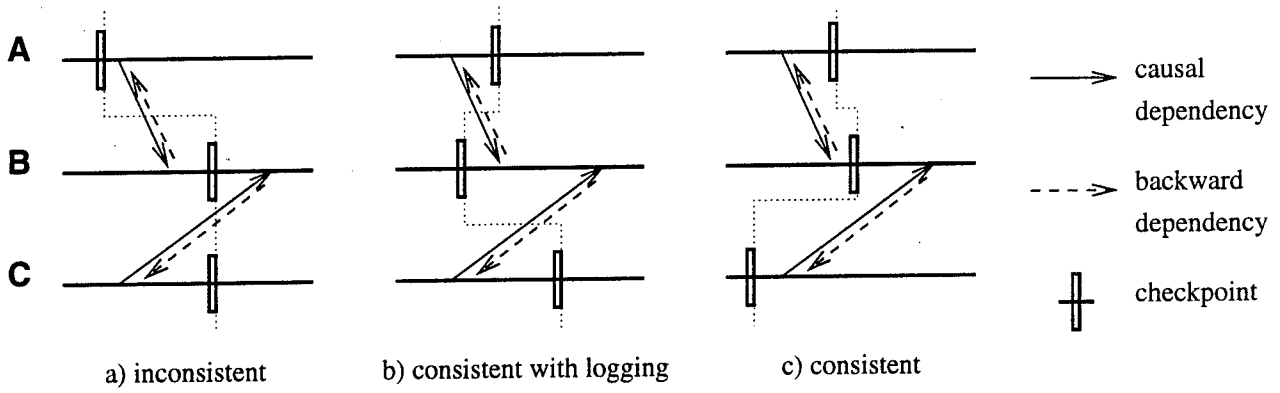


Figure 1: Consistent and inconsistent global checkpoints.

and inconsistent global checkpoints. In Figure 1a, there is a causal dependency from after the checkpoint on process A to before the checkpoint on process B, so the global checkpoint is inconsistent. In Figure 1b, there are no causal dependencies from after the global checkpoint to before, so it is consistent if messages can be replayed from a log. In Figure 1c, all dependencies begin and end before the global checkpoint, so it is consistent with or without logging.

Our passive server model for DSM systems is derived from the message-passing model. We model program execution in DSM systems as a set of client processes which run the application program and a set of passive server processes which provide a shared-memory image to the clients. The servers are considered passive since they only change state due to interaction with a client. In the clients, events can be either internal events, read events, or write events. Internal events only depend on and affect the local state of the process. Read events send a read request to a local server, wait for a reply with the value of a data item and then update their local state with the value. Write events send a write request with a value to a local server and wait for a reply. Events in servers are always triggered by the receipt of a request message, either from a client or another server. Request messages are handled by the servers in FIFO order. After the request message is received, server events may send and receive additional messages.

A write or read event in a client, together with the events it causes in the servers may be collectively called an interaction. The passive server model differs from the message-passing model in that it collects all the events in a process during an interaction into one single event. Figure 2 illustrates a typical write interaction in DSM in terms of the passive server model. Since events are defined as atomic actions, a system only conforms

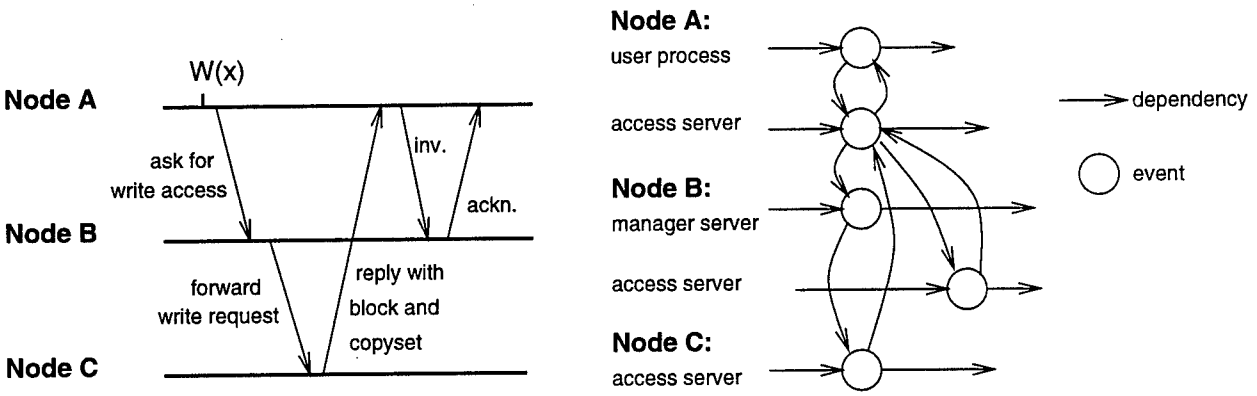


Figure 2: Typical write interaction in the passive server model.

to the passive server model if it is possible to guarantee the atomicity of interactions. Checkpointing events are inserted as in the message-passing model. Note that checkpoints should be constrained to occur outside interactions, so that events remain atomic. The passive server model contains all the dependencies of the message-passing model. However, we shall show that its structure allows many of these dependencies to be eliminated.

4 Design of a Recoverable DSM

We now outline the high-level design of a recoverable DSM system that conforms to the passive server model. We use a fixed distributed manager (FDM) algorithm for maintaining coherence [31], which is general enough to encompass many software and hardware based DSM designs [16, 9, 21, 29]. We use a simple method to ensure that interactions are atomic, as prescribed in the passive server model. We introduce a recovery scheme where the only directory information needed to correctly recover a block is an ownership timestamp maintained by every node. Our design eliminates most message-passing dependencies, leaving only those dependencies due to transfers of blocks of data. Our design can be implemented on a hardware DSM by replacing the directory by a custom recoverable directory. No other changes need to be made to the hardware.

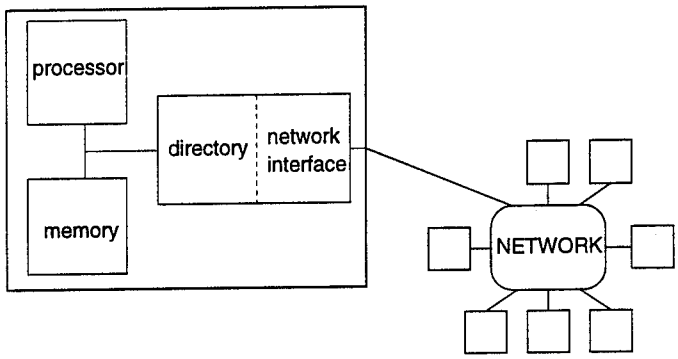


Figure 3: Conceptual organization of DSM processing node.

4.1 Basic system model

Our target DSM system consists of an arbitrary number of processing nodes connected by a general purpose interconnection network. As illustrated in Figure 3, conceptually a processing node consists of a processor and memory connected to the network through a remote data interface. The remote data interface is responsible both for communicating with other processing nodes, and keeping its own node's shared data coherent by keeping a directory of remote blocks. In a hardware implementation, the directory usually keeps track of cache blocks by snooping on the system bus. In a software implementation (shared virtual memory), the directory is part of the virtual memory's page table.

In the FDM algorithm, every memory block is assigned to a home node. Any request for access to a block is sent to the home node, which then forwards the request to the owner of the block. In the passive server model, every node contains one or more user processes, and a local server process for every block of shared data. In addition, for every block for which a node serves as the home node, it contains a manager server process. To simplify our treatment, unless otherwise noted, we combine all the local block server processes on a node into one access server. Likewise, all manager servers on a node are combined into one manager server process. Since we are concerned with inter-node communication, we do not need to consider the messages between the user processes and their local access server. These simplifications restrict us to considering only recovery schemes that roll back all the activities of the combined servers as a unit.

The interactions that can occur in the FDM coherence scheme are illustrated in terms of the simplified passive server model in Figure 4. Interactions are initiated by a read or write fault on a user process. An

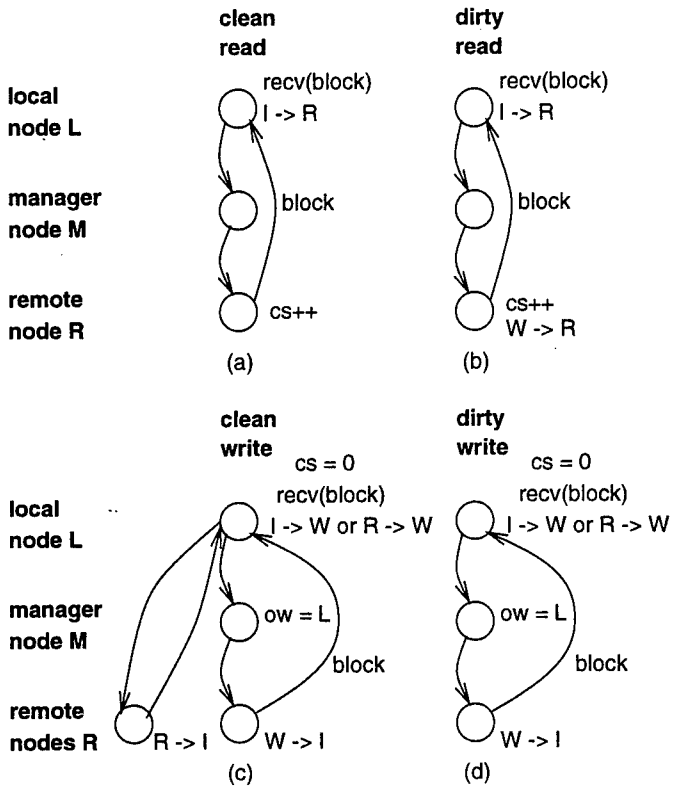


Figure 4: Possible interactions during accesses in a fixed distributed manager DSM.

interaction consists of an event in each server and the dependencies between event. Next to the graphical representation of an event, the state changes of that event are shown. A node can either have writable access (W), read-only access (R), or no access (I) to a block. When a node gains writable access to a block, it sends invalidations to all other nodes that have a copy. A copyset (indicated by "cs" in the figure) keeps track of other nodes with a copy to eliminate unnecessary invalidations. In a system with limited cache space per node, any of the interactions may include an additional writeback of a replaced block to its home node. Only causal dependencies are shown in the figure; for every causal dependency there is a corresponding backward dependency. Next we show that, by carefully constructing our recoverable DSM system, we can eliminate all backward dependencies and many causal dependencies.

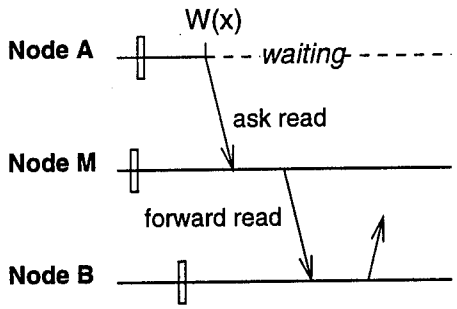


Figure 5: Situation resulting from an incomplete request.

4.2 Maintaining atomicity of server events

The passive server model treats a server's whole response to a request message as an atomic event to be able to avoid issues of deadlock and spurious messages caused by partially completed interactions. Consider the situation in Figure 5, where a request for write access from node A has been forwarded, by the accesses block's manager on node M, to node B. If Node B or Node C rolls back to its previous checkpoint while handling the request, the interaction is aborted and Node A will wait indefinitely for the reply from Node B. If Node A rolls back while waiting, it will receive an unexpected reply message from B after the rollback. Analogous situations would occur if one node rolled back to a checkpoint inside the interaction, while the others continued. In the message-passing model, consistency is guaranteed by the dependencies introduced by the messages from dependencies from node A to the manager node M and from node M to node B. However, we would like to be able to eliminate these dependencies in our DSM system.

To maintain atomicity, the system should never roll back to a global state with partially completed events. It is simple to constrain checkpoints to occur only outside of interactions, so that a node cannot roll back to a state inside an interaction. It is not possible to delay a rollback until the end of an interaction, however. A request numbering scheme can be used to handle spurious messages. In the example of Figure 5, if Node A assigns a unique number to the request message, and sequence number is propagated through the requests to the reply message, then if Node A rolls back, it can reject the reply as spurious. The simplest way to avoid the deadlock situations is to always roll back a node if it is waiting for a reply when it receives notice that another node has rolled back. This scheme can be improved by coding the request interaction so no state is permanently saved until after the last reply has been received. Then the request can simply be aborted,

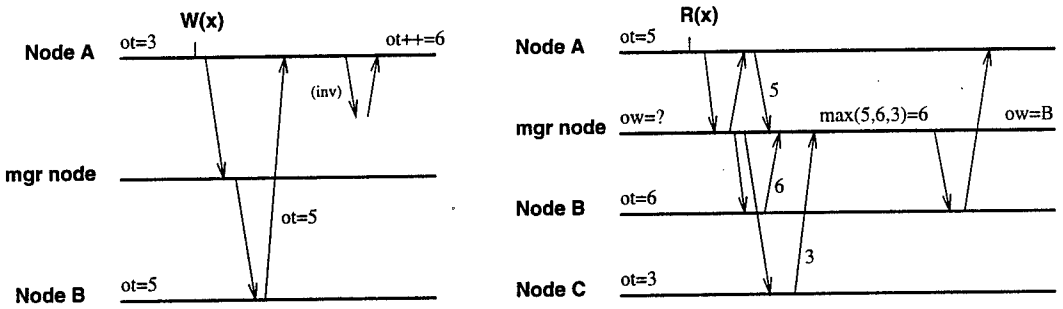
avoiding a rollback. To avoid that nodes not involved in an interrupted interaction abort, nodes waiting for a reply can instead continue to wait for a fixed amount of time, and abort if a reply is not received. For example, if Node B rolls back before sending the reply, all other nodes in the system that are waiting for a reply would set a timer. In the absence of other rollbacks, all nodes except A would probably receive the reply before timing out. Node A would time out, abort, and retry the request. Note that since interactions are relatively short events, most of the time it will not be necessary to abort any requests to maintain atomic execution.

4.3 Maintaining ownership timestamps

As long as concurrent writes by two nodes to the same block are not allowed, all modifications to any block in a DSM system can be totally ordered. The node that modified a block last is considered its owner and supplies a copy of that block to any other node that needs access. When one or more nodes roll back, the node that had ownership of a block latest in time before the global checkpoint should receive ownership after rollback. Many of the dependencies between events in an interaction are there to maintain this ownership consistency.

By implementing an ownership timestamp scheme where every node keeps track of the last time it became owner of a block, we can eliminate these dependencies. Furthermore, the scheme allows all directory information besides the ownership timestamps to be lost after rollback without affecting correct execution. As illustrated in the example of Figure 6a, every time ownership of a block is transferred, the old owner sends its current value of that block's ownership timestamp to the new owner. Upon receiving the value, the new owner increments it, and then stores it as its ownership timestamp for the block. This procedure guarantees that the current owner of a block always possesses the largest ownership timestamp for that block. Periodically, when the timestamp overflows, all nodes need to synchronize to reset their timestamps to ensure correct ordering.

As long as the system maintains ownership timestamps, no other block state information is needed for correct operation. If no other ownership information is available, on a miss to a block the owner can be found by gathering all ownership timestamps from all nodes, and selecting the largest. As long as a node maintains a record of the blocks it is certain it owns, erroneous ownership information can be tolerated. A request for



a) transfer of ownership

b) recovery of ownership information on manager

Figure 6: Maintaining ownership timestamps during ownership transfers and during recovery.

access to a block that might not be owned by a node is rejected, and the true owner is found by using the ownership timestamps. The copyset maintained by the owner of every block is an optimization which limits the number of invalidations that need to be sent out, but is not necessary for correctness. Furthermore, the copyset on a node may contain a superset of all the other nodes that actually have a copy of a block. The only consequence is that unneeded invalidations are sent to some nodes. At a high performance penalty, even the access right information can be eliminated without affecting correctness by treating every access to a shared block as a miss.

To make a DSM system recoverable, only the ownership timestamps and the data in all blocks are checkpointed. After rollback, all information about the state of a block is set to unknown. As shown in the example of Figure 6b, the first access to a block after rollback recovers its directory information. When a block with an unknown owner is first accessed, the ownership timestamps are used to determine the owner. Ownership information is updated so that further accesses do not need to use the ownership timestamps. When exclusive write access is needed to a block with an unknown copyset, all other nodes are sent invalidates and the copyset is changed from unknown to empty. Gradually, all ownership and copyset information is updated, and the original DSM algorithm is used for further accesses.

4.4 Eliminating dependencies

We now analyze the FDM algorithm to ascertain which dependencies can be eliminated when using ownership timestamps and the passive server model. Consider the role of the manager server in an interaction. On a read interaction, the manager merely forwards the request. The state on the manager's node (node M) is the same before the interaction as afterwards. If node M rolls back, the ownership information it maintains is lost, but it can be recovered by comparing the ownership timestamps on all nodes. Therefore all dependencies involving node M in a read interaction can be eliminated. In a remote write interaction, node M changes state; it records the new owner of the block. If node M rolls back, it loses ownership information, so the timestamps are used to find the owner. If another node rolls back, node M may contain erroneous ownership information. Any request that is routed to the wrong owner by M will be rejected however, and the timestamps will be used to find the correct owner. So again we can ignore all dependencies with node M. Therefore, by using ownership timestamps, the function of the manager has been made redundant, and does not have to be considered for rollback to a consistent state.

Having eliminated the dependencies with the node that contains a block's manager, we can now analyze interactions solely in terms of the dependence between the local (L) and remote (R) nodes. In a remote read access to a clean block (Figure 4a), the state of node R is not affected by the interaction, except for the addition of a member to the copyset. When node L rolls back, the state of the recovery line is the same as if node R also rolled back, except for the extra member of the copyset. Since the copyset is allowed to be a superset of all the nodes that have readable copies, the recovery line is consistent. So the backward dependency $L \xrightarrow{B} R$ can be eliminated. When the remote read access is to a dirty block (Figure 4b), a rollback of node L will cause a situation where node R has lost write permission without guaranteeing that a copy of the dirty block has been saved on another node. However, node R is still the owner, so any further requests will be supplied from its copy of the block. Therefore the dependency $L \xrightarrow{B} R$ can again be eliminated. So, on a remote read, there remains only the causal dependency, $R \xrightarrow{C} L$, from the remote node to the local node.

Next, we consider a remote write access (Figure 4c, 4d). Ignoring invalidations, the interactions for a clean and dirty write are identical, with the access permission of the block on the remote node changing from writable to read-only. If node L rolls back and re-executes the write access, the request is directed by

Table 1: Address trace characteristics.

program	description	tot. num. of references	data reads		data writes	
			total	shared	total	shared
gravsim	N-body simulator	92,178,814	33,266,880	12,484,455	6,392,078	251,694
fsim	fault simulator	149,918,375	50,950,933	39,326,911	3,958,919	999,127
tgen	test generator	101,264,382	32,613,809	16,550,450	4,461,889	642,796
pace	circuit extractor	87,861,165	23,266,576	1,286,787	7,842,338	348,524
phigure	global router	132,998,231	38,244,233	4,281,207	11,530,981	1,876,400

the manager to node R. Node R rejects the request because it has given up ownership. This rejection will cause the ownership timestamps to be used to find the correct copy of the block. Therefore the dependency $L \xrightarrow{B} R$ is eliminated. The causal dependency $R \xrightarrow{C} L$ remains since it transmits a block of data.

If the block is readable by more than one remote node when the local node asks for write access, all the copies in the remote nodes will be invalidated. A node L can safely roll back past an interaction in which it invalidated node R'. In the global state after rollback, it will appear as if node R' has been invalidated spontaneously and at the next access node R' can ask the owner for a new copy of the block. Therefore there is no dependency $L \longrightarrow R'$. If node R' rolls back past the interaction, the access rights of all its blocks are set to unknown. Therefore any access to a block that was invalidated before rollback will ask the owner for a new copy, just as if the block had been invalidated. So the remaining $R' \longrightarrow L$ dependency is eliminated, resulting in a dependence-free invalidation interaction.

Figure 7 shows the dependencies eliminated using the passive server model and ownership timestamps. All backward dependencies, as well as all dependencies due to invalidations are eliminated. The only dependencies remaining are those due to data being transferred to the requester on a remote access.

To determine the reduction in dependencies caused by using our scheme we performed trace-driven simulations with multiprocessor address traces from five parallel scientific programs running on an Encore Multimax. The traces were generated by the TRAPEDS address tracer from execution on seven processors. Each trace contains at least 10 million memory references per processor [39]. Table 1 describes the characteristics of the traces used.

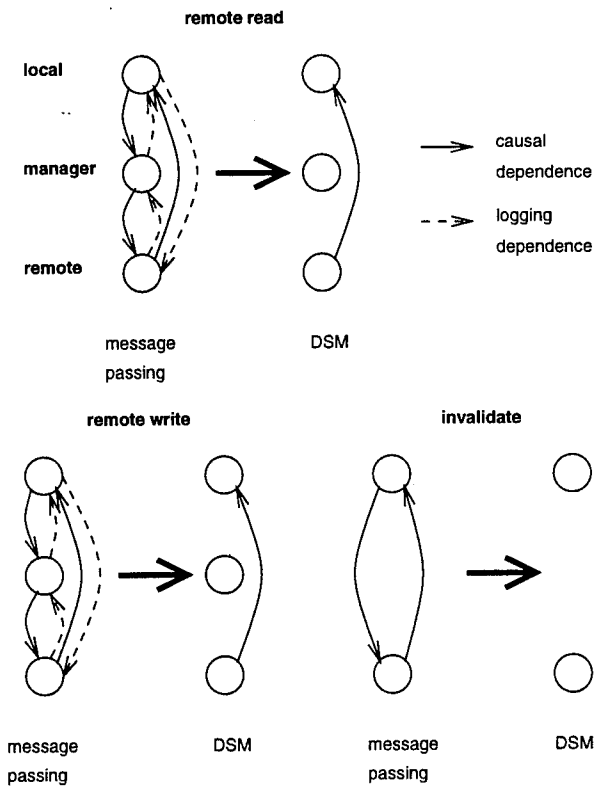


Figure 7: Reduced dependencies.

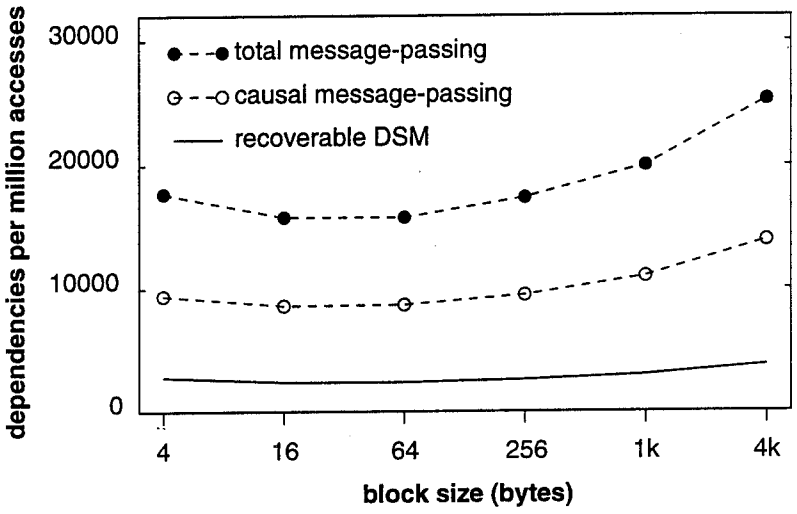


Figure 8: Frequency of dependency comparison

Figure 8 presents the frequency of dependencies in the message-passing model and in the DSM passive server model using ownership timestamps. Our DSM model has more than a sixfold decrease in dependencies over the message passing model. An implementation using this model therefore significantly reduces dependency tracking overhead and/or the probability of rollback propagation. The causal dependencies for the message-passing model are also plotted in the Figure. This is the number of dependencies that would need to be considered in the message-passing model if logging is implemented. In the DSM model, all backward dependencies are eliminated, so logging need never be used. Even then, there still is a decrease of these causal dependencies by a factor of 3.5.

4.5 Implementation issues

The high-level design described above can be implemented directly in a shared virtual memory implementation of DSM on a network of workstations. The DSM coherency protocol needs to be augmented to allow broadcast for ownership timestamps in case the manager has lost ownership information. To allow rollback recovery, it is necessary to have the ability to checkpoint the state of the CPU and the virtual memory to stable storage. Various software methods exist to achieve this [14, 32]. The only enhancement needed is the abil-

ity to also checkpoint and restore ownership timestamps. Depending on whether coordinated or independent checkpointing is chosen, checkpoint coordination [14, 30] or dependency tracking [37] algorithms also have to be implemented.

When specialized hardware is used to implement DSM, several additional issues need to be considered when implementing rollback recovery. In hardware-based DSM systems, memory coherence is maintained at the cache block level [9, 21, 29]. We map our passive server model onto such systems by considering the directory/communications hardware as implementing both the access server and manager server for a node, and modeling the rest of the node as one monolithic user process. Note that even in systems such as DASH [29], where a nodes consist of a collection of multiple processors, caches, and memory, we still model all the components besides the directory as one user process.

In hardware DSM, checkpointing can still be performed by software. All other functions can be integrated in a custom directory controller. No other modifications to the node hardware are necessary. Inter-node communication occurs between caches through the directory, allowing it to perform any required dependency tracking and coordination functions. Likewise, the extension to the protocol to allow operation with ownership timestamps is implemented in the directory. The directory is responsible for saving ownership timestamps during checkpointing and restoring them after rollback.

One additional complication in hardware DSM is the necessity to checkpoint all the data maintained on the processing node, which includes processor registers, processor cache, and main memory. All the processor internal state, including registers, are be simply flushed to the main memory via software before checkpointing. The coherency mechanism between the processor cache and the I/O controller ensures that the contents of all memory that resides locally will be saved on a checkpoint, even if the contents reside in a dirty state in the cache. However, there may also be blocks present in the cache which are managed by remote nodes. To avoid having to checkpoint these blocks, they can be flushed to their respective home nodes before a checkpoint is taken. This solution, however, causes a large amount of traffic on the network before every checkpoint. To avoid this overhead, the directory controller needs to be modified to intercept these flushes, and store their contents in a reserved portion of the local virtual memory, where they are subsequently saved as part of the checkpoint. After a rollback the directory controller again has to intervene, to restore the con-

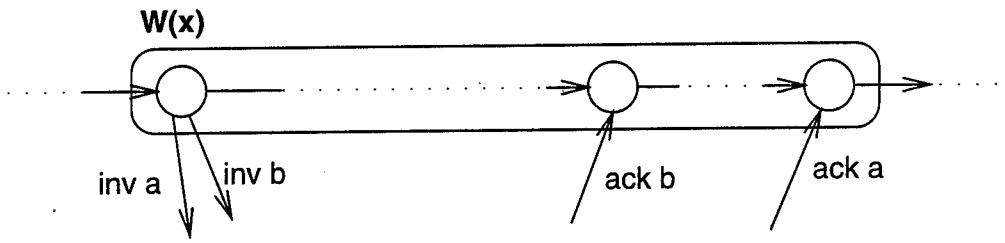


Figure 9: Atomic event between write initiation and last acknowledgement.

tents of the affected blocks.

To reduce the performance impact of remote memory access latency in a DSM system, a relaxed memory consistency model can be employed [1, 19]. In the next section we describe a design that, by using lazy release consistency, reduces further the dependencies that need to be considered for correct rollback. Here we show that the release consistency model used in DASH, where invalidate acknowledgements can be delayed, fits the passive server model, and therefore can use the shared-memory dependencies of the FDM algorithm. In DASH, a write interaction can complete before it receives all the acknowledgements from the invalidates it sent out. While invalidate acknowledgements to a block are pending, all other accesses from other nodes to the block are delayed. Likewise, certain special *release* interactions on the local node are also delayed. To allow release consistency in the passive server model, it is necessary to model the individual blocks in the access server as separate processes. Since all requests for access to a block are denied while an invalidate is pending, and accesses to other blocks are handled by other block access servers, it is now possible to extend the write interaction by considering the period from the initiation of the write access to the receipt of the last acknowledgement as one event (see Figure 9). Therefore, checkpointing cannot occur while an invalidate is pending, and a node with pending invalidates may have to roll back if another node that has not yet replied rolls back.

5 Recoverable DSM with Lazy Release Consistency

In software DSM systems, there is a high overhead for every message sent between nodes. To reduce the number of messages needed to maintain consistency, lazy release consistency [26] has been developed. The

model is successful in reducing overhead of maintaining consistency, approaching the performance of a bus-based multiprocessor in a software-based implementation on an asynchronous transfer mode (ATM) network of workstations [11]. The lazy release consistency model has the additional advantage that inter-node communication can be restricted to synchronization accesses, thereby dramatically reducing the number of dependencies needed to assure correct rollback recovery.

5.1 Memory consistency models and rollback

The consistency model traditionally used in shared memory systems, sequential consistency, guarantees that all memory accesses appear to execute atomically and in program order [28]. The model was developed to reflect the programmer's intuitive understanding of the correct execution of a multiprocessor. If sequential consistency is not maintained, synchronization and mutual exclusion using loads and stores to lock variables will not necessarily function correctly. Lamport defined sequential consistency as follows [28]: "[A system is sequentially consistent if] the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."

One can view the dependencies needed to ensure correct rollback in a traditional system as those necessary to maintain sequential consistency. For example, in Figure 10 sequential consistency is violated by allowing node A to roll back past a dependency with node B. Node A sets variable $x = 0$, followed by $x = 1$ some time later. Then node B reads x twice, storing the results in a and b. In the time between the two reads on node B, A rolls back to the checkpoint taken before the second write. On node B, the second read of x occurs before the rolled-back node A reassigns $x = 1$. The outcome is that $a = 1$ and $b = 0$, which, if all operations were executed in sequential order, is only possible if the assignment $x = 1$ came before $x = 0$. This is not the order specified by the program on node A, therefore sequential consistency is violated.

The strong sequential consistency requirements prevent systems designers from making certain performance optimizations. By using a more relaxed consistency model, accesses to remote memory locations can be delayed and reordered, increasing performance. Furthermore, it is possible to define a relaxed memory consistency model so that any system using it appears sequentially consistent to most programs. The most

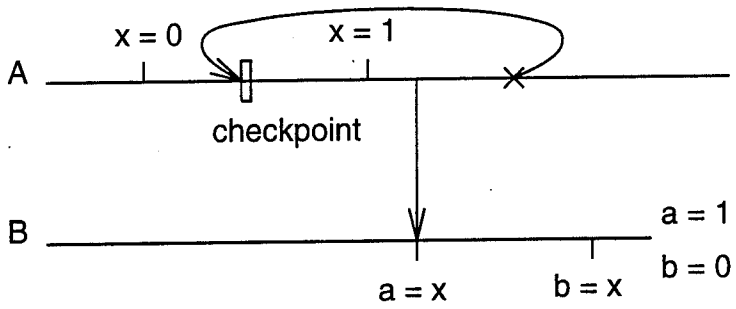
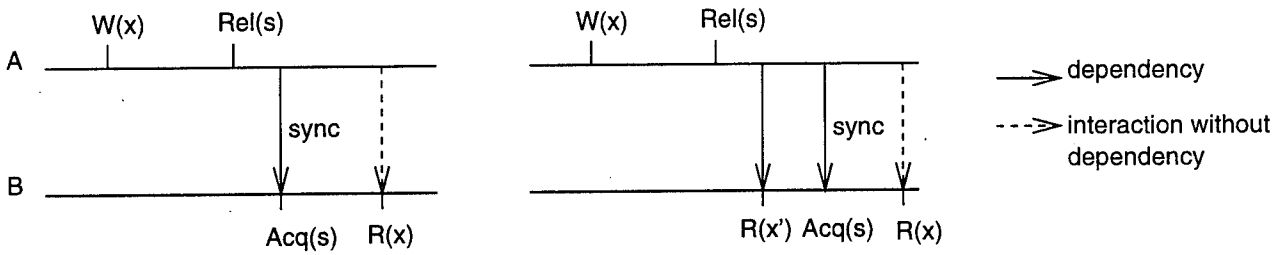


Figure 10: A violation of sequential consistency caused by a rollback past the read of a shared variable by a remote node.

popular relaxed consistency model is release consistency [19, 26], where the *happens-before-1* relation [1] is enforced. In the happens-before-1 relation, intra-processor data accesses are ordered by program order, and inter-processor accesses are ordered by pairs of release and acquire accesses to synchronization variables. For example, if processor A unlocks lock variable s and processor B later locks s, then all data accesses before the unlock on A are ordered before all data accesses after the lock on B. As long as the application programmer ensures that the program does not have any data races, a release consistent system is indistinguishable from a sequentially consistent system. A data race occurs when two data operations access the same memory location, they are not both reads, and they are not ordered by happens-before-1.

The conditions that allow a release consistency system to appear sequentially consistent also allow certain dependencies to be removed from those necessary for correct rollback. Consider the example in Figure 11a. Memory location x is written on processing node A. It is now guaranteed that node B will not read this location, until node A releases (unlocks) synchronization variable s and node B acquires (locks) it. So there will be no dependencies due to the read of variable x until the acquire. When node B does read variable x, it is guaranteed to read the value written on A before the release, since node A may not write x after the release. Therefore, the dependency due to the read of x on B is covered by the dependency due to the acquire and can be removed. If node A rolls back to a point after the release, it does not change the value of x, so node B does not have to roll back. Therefore, the only dependency is due to the acquire of variable s.

If coherence is maintained separately for every memory location, the only dependencies in a data-race-free system are those due to acquires of variables. Unfortunately, when coherence is maintained for blocks of memory locations, false sharing will introduce additional dependencies [17]. In Figure 11b, the situation



a) dependency due to data read is covered by acquire dependency

b) dependency due to false sharing read is not covered by acquire dependency

Figure 11: Dependencies in data-race-free programs.

is the same as in the previous example, except a read of location x', which shares a block with x, occurs on B before the acquire. Since variable x is now known to node B before the acquire, it causes a dependency. Therefore, the data-race-free condition is not enough to restrict dependencies to synchronization accesses. However, for data-race-free programs, a relaxed memory consistency model can be implemented that appears sequentially consistent, and does restrict dependencies to synchronization interactions.

5.2 Design of recoverable DSM with lazy release consistency

To guarantee correct rollback while only tracking dependencies due to synchronization accesses, it is necessary to eliminate false sharing. Multiple writer protocols are used in software distributed shared memory systems to eliminate excessive transfer of blocks due to false sharing [7]. In a multiple writer protocol, two or more nodes can simultaneously update their local copy of a shared block. For every write, a record describing the modification (called a *diff*) is created. Only these diffs are propagated to other nodes, guaranteeing that the contents of locations in a block that have not been updated do not need to be transmitted.

To further reduce message traffic in DSM systems, lazy release consistency (LRC) with a multiple writer protocol has been implemented [26, 13]. LRC directly implements the happens-before-1 relation by using vector timestamps. Execution on processors is divided into intervals by synchronization accesses. Every processor keeps track of which intervals it is aware by updating its vector timestamp on any interaction. On an acquire, the vector timestamps are used to propagate *write notices* of all modifications to memory locations that occurred before the acquire in the happens-before-1 order. Received write notices have to be stored lo-

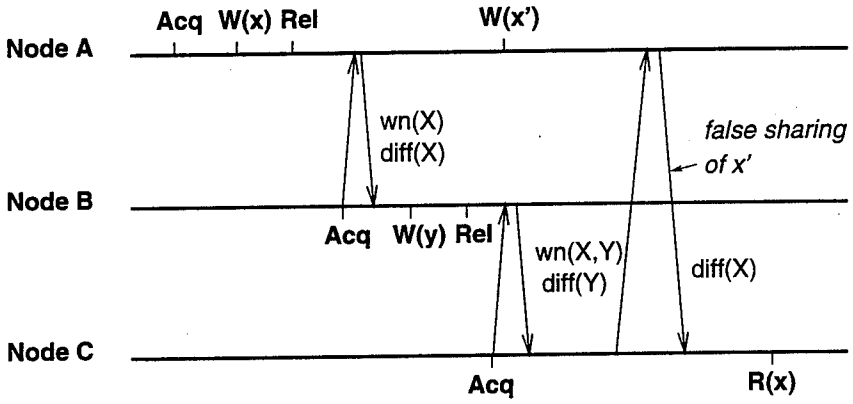
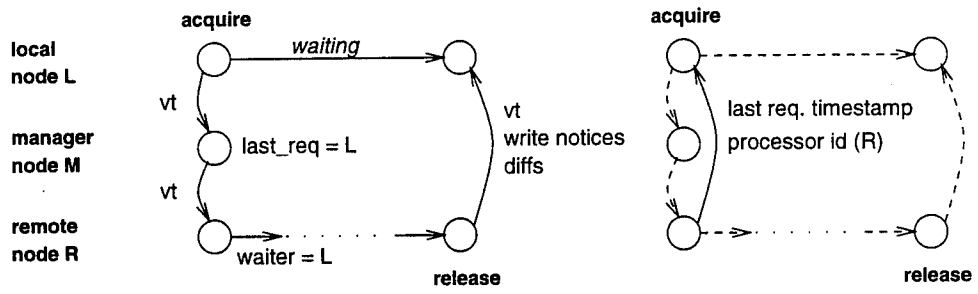


Figure 12: False sharing in LRC algorithm.

cally, in case they need to be propagated to another node. Periodic garbage collection deletes write notices have been propagated to all nodes. There is no concept of ownership of memory blocks; all the information on the contents of pages is transferred directly from the releaser to the acquirer of a lock. When an update strategy is used with LRC, interaction between processors, and therefore dependencies, only occur at acquires.

Even in LRC with multiple writers and an update protocol [13], however, there is a potential dependency with every other node at every acquire of the lock. Consider the example in Figure 12. Processor A writes to location x in block X, then node B writes to location y in block Y, and finally node C reads location x. All updates are ordered by one lock. Updates, in the form of diffs, are transmitted at the time of the acquires of the lock. In the original protocol [13, 26], when node C acquires the lock, it receives write notices for blocks X and Y, but only receives a diff for block Y. It has to fetch the diffs for block X from its last writer, node A. If node A writes a variable x' in block X before the interaction, its new value will be propagated. Therefore there is still false sharing of location x' between nodes A and C. To eliminate all false sharing, it is necessary to modify the protocol to maintain diffs together with their corresponding write notices. In the example, the new value of x would be received by node C from node B with the write notice for page X, and there would be no interaction with node A.

In LRC, since locks do have to be sequentially consistent with each other, they are managed separately from data blocks. Every lock has a statically assigned manager. This manager keeps a *last_req* record of the node that most recently requested the lock. When a new request comes in from an acquire, it forwards it to the



a) acquire interaction in LRC algorithm

b) extra message to support rollback recovery

Figure 13: Acquire interaction in LRC algorithm.

node specified in last_req, and then changes last_req to the new requester. In our algorithm, the only interaction occurs on an acquire. The interaction is shown in Figure 13a. The requester sends its vector timestamp (vt) to the manager for the lock, which forwards it to the node that last held the lock. If this node has not released the lock, the request is delayed until the release. Otherwise the request is fielded immediately. The releaser uses the received vector timestamp to decide which write notices to send to the acquirer. All write notices, their corresponding diffs, and the requester's vector timestamp are then sent to the acquirer, which incorporates this information in its directory before continuing.

In our LRC checkpointing and rollback algorithm, most of state of the node, including all memory blocks, write notices, and diffs are checkpointed. As described below, however, the last_req records in the lock managers, and any record of a waiter at a lock that has not been released are not checkpointed and restored. To conform to the passive server model, the acquire interaction needs to be atomic. In the previously described scheme for maintaining atomicity of interaction events, any node that is waiting for a reply rolls back when it becomes aware of a rollback in the system. This method is not practical in the LRC case, since an acquire may be waiting for a long time while another node holds its lock. To make the acquire interaction atomic, the node holding the lock replies with its id when it first receives the acquire request. (see Figure 13b). The acquire request is recorded by the receiving node but is not saved as part of any checkpoint. If the requester node receives notice that the node holding the lock has rolled back and lost the acquire request, the requester also rolls back. This scheme ensures that all nodes waiting on an acquire do not wait indefinitely, but re-send their request if the target node rolls back.

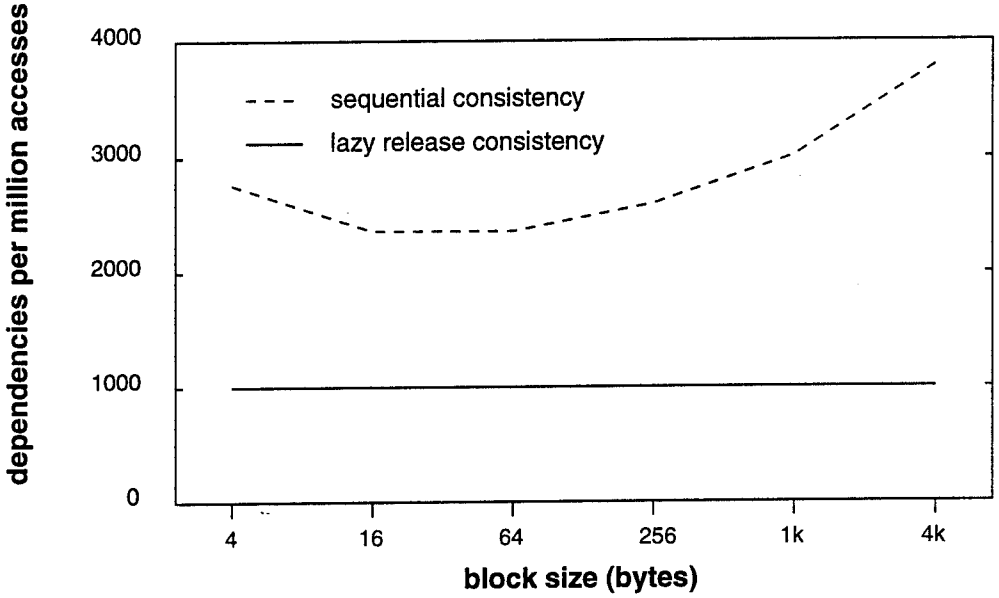


Figure 14: Dependency frequencies with different memory consistency models.

To eliminate the dependencies with the node managing the lock, a last requester timestamp, similar to the ownership timestamp described earlier, is implemented. As shown in Figure 13b, when a node receives an acquire request, it sends its last requester timestamp and its node id to the new requester. The last requester information maintained by the lock managers is not checkpointed, and after rollback, broadcast for the last requester timestamps is used to recreate the record. Since the lock manager is now redundant after rollback, only the dependencies between the acquiring node and the releasing node remain. There is obviously a causal dependency from the releaser to the acquirer when it sends the write notices. The backward dependency also remains.

By only communicating during synchronization, our rollback scheme for LRC significantly reduces the number of dependencies that need to be considered. Figure 14 shows the results of simulations with the shared-memory address traces for sequential consistency and lazy release consistency. With lazy release consistency, there are two dependencies, one causal, and one backward, per acquire. The backward dependencies can be eliminated if the release messages are logged. The number of acquires, and therefore the frequency of dependencies, does not depend on the block size. From our simulations, the frequency of dependencies with lazy release consistency is about 1000 per million accesses. This compares, for 64-byte blocks, to a frequency of 2400 for sequential consistency, and a frequency of about 10000 if every message causes a dependency.

6 Conclusions

Checkpointing and rollback recovery in distributed message passing systems is a mature area of research. In a message passing system, every message causes a dependency from the sender of the receiver. In a shared memory model, even if implemented on top of message-passing, some of the messages do not need to cause dependencies. Previous work, both in checkpointing [3, 44, 43] and distributed debugging [20, 33] has assumed a less strict dependency pattern for shared memory than message passing. By using the passive server model, our work shows that the dependency pattern for shared memory can be derived from that for message passing, and therefore can be used in architectures where a shared memory image is provided via physically distributed memory.

We used the FDM algorithm to show that only the dependencies due to the direct sharing of memory blocks need to be considered if a recoverable DSM is designed so it can tolerate the loss of ownership information in a block's home directory. Similar analysis can be performed on other algorithms for maintaining consistency. Lazy release consistency with multiple owners can be used to further limit dependency overhead to acquire/release interactions. However, its complexity is too high to be implemented in hardware DSM. In software DSM, where messaging overhead is higher, LRC increases performance, even without checkpointing. In such systems, the designer of a rollback recovery scheme can take advantage of the reduced number of dependencies to decrease overhead.

Acknowledgements

Our work benefited from discussions with Alain Gefflaut at IRISA, and Gaurav Suri, Yi-Min Wang and Sujoy Basu at Illinois.

References

[1] S. V. Adve and M. D. Hill, "A unified formalization of four shared-memory models," *IEEE Trans. on Parallel and Distributed Systems,* Vol. 4, No. 6, pp. 613–624.

[2] R. E. Ahmed, R. C. Frazier, and P. N. Marinos, "Cache-aided rollback error recovery (CARER) algorithms for shared-memory multiprocessor systems," *Proc. 20th Int. Symp. on Fault-Tolerant Computing,* 1990, pp. 82–88.

[3] M. Banâtre, A. Gefflaut, P. Joubert, P. Lee, and C. Morin, "An architecture for tolerating processor failures in shared-memory multiprocessors," Tech. Report 707, IRISA, Rennes, France, Mar. 1993.

[4] P. A. Bernstein, "Sequoia: a fault-tolerant tightly coupled multiprocessor for transaction processing," *Computer,* Vol. 21, No. 2, Feb. 1988, pp. 37–45.

[5] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle, "Fault tolerance under UNIX," *ACM Trans. on Computer Systems,* Vol. 7, No. 1, Feb. 1989, pp. 1–24.

[6] L. Borrmann and M. Herdieckerhoff, "A coherency model for virtually shared memory," *Proc. Int. Conf. on Parallel Processing,* 1990, pp. II-252–II-257.

[7] J. B. Carter, J. K. Bennett, and W. Zwaenepoel, "Implementation and performance of Munin," *Proc. 13th ACM Symp. on Operating Systems Principles,* 1991, pp. 152–164.

[8] K. M. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM Trans. on Computer Systems,* Vol. 3, No. 1, Feb. 1985, pp. 63–75.

[9] D. Chaiken and A. Agarwal, "Software-extended coherent shared memory: performance and cost," *Proc. 21st Int. Symp. on Computer Architecture,* Apr. 1994, pp. 314–324.

[10] M. Clarke, "MPP comes to the desktop," *Electronic Engineering Times,* 5 Sep. 1994, pp. 1, 37.

[11] A. L. Cox *et al.,* "Software versus hardware shared-memory implementations: a case study," *Proc. 21st Int. Symp. on Computer Architecture,* Apr. 1994, pp. 106–117.

[12] F. Cristian and F. Jahanian, "A timestamp-based checkpointing protocol for long-lived distributed computations," *Proc. 10th Symp. on Reliable Distributed Systems,* 1991, pp. 12–20.

[13] S. Dwarkadas, P. Keleher, A. L. Cox, and W. Zwaenepoel, "Evaluation of release consistent software distributed shared memory on emerging network technology," *Proc. 20th Int. Symp. on Computer Architecture,* May 1993, pp. 144–155.

[14] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel, "The performance of consistent checkpointing," *Proc. 11th Symp. on Reliable Distributed Systems,* 1992, pp. 39–47.

[15] M. J. Feeley, J. S. Chase, V. Narasayya, and H. M. Levy, "Integrating coherency and recovery in distributed systems," *Proc. Symp. on Operating Systems Design and Implementation,* Nov. 1994.

[16] B. D. Fleisch and G. J. Popek, "Mirage: a coherent distributed shared memory design," *Proc. 12th ACM Symp. on Operating Systems Principles,* Dec. 1989, pp. 211–223.

[17] A. Gefflaut, personal communication.

[18] A. Gefflaut, C. Morin, and M. Banâtre, "Tolerating node failures in cache only memory architectures," *Proc. Supercomputing '94,* Nov. 1994.

[19] K. Gharachorloo *et al.*, "Memory consistency and event ordering in scalable shared-memory multiprocessors," *Proc. 17th Int. Symp. on Computer Architecture,* 1990, pp. 15–26.

[20] L. Gunaseelan and R. J. LeBlanc, "Event ordering in a shared memory distributed system," *Proc. 13th Int. Conf. on Distributed Computing Systems,* 1993, pp 256–263.

[21] M. D. Hill, J. R. Larus, S. K. Reinhardt, D. A. Wood, "Cooperative shared memory: software and hardware support for scalable multiprocessors," *Proc. 5th Int. Conf on Architectural Support for Programming Languages and Operating Systems,* Oct. 1992, pp. 262–273.

[22] R. Jalili, F. Heskens, D. M. Koch, and J. Rosenberg, "Operating system support for object dependencies in persistent object stores," *Proc. Workshop on Object-oriented Real-time Dependable Systems,* Oct. 1994.

[23] G. Janakiraman and Y. Tamir, "Coordinated checkpointing-rollback error recovery for distributed shared memory multicomputers," *Proc. 13th Symp. on Reliable Distributed Systems,* Oct. 1994, pp. 42-51.

[24] B. Janssens and W. K. Fuchs, "Relaxing consistency in recoverable distributed shared memory," *Proc. 23rd Int. Symp. on Fault-Tolerant Computing,* Jun. 1993, pp. 155-163.

[25] B. Janssens and W. K. Fuchs, "Reducing Interprocessor Dependence in Recoverable Distributed Shared Memory," *Proc. 13th Symp. on Reliable Distributed Systems,* Oct. 1994, pp. 34–41.

[26] P. Keleher, A. L. Cox, and W. Zwaenepoel, "Lazy release consistency for software distributed shared memory," *Proc. 19th Int. Symp. on Computer Architecture,* 1992, pp. 13–21.

[27] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Trans. on Software Engineering,* Vol. SE-13, No. 1, Jan. 1987, pp. 23–31.

[28] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. on Computers,* Vol C-28, No. 9, Sep. 1979, pp. 690–691.

[29] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, J. Hennessy, "The directory-based cache coherence protocol for the DASH multiprocessor," *Proc. 17th Int. Symp on Computer Architecture,* 1990, pp. 148–159.

[30] K. Li, J. F. Naughton, and J. S. Plank, "Checkpointing multicomputer applications," *Proc. 10th Symp. on Reliable Distributed Systems,* 1991, pp. 1–10.

[31] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Trans. on Computer Systems,* Vol. 7, No. 4, Nov. 1989, pp. 321–359.

[32] M. Litzkow and M. Solomon, "Supporting checkpointing and process migration outside the UNIX kernel," *Proc. Usenix Winter Conf.,* 1992.

[33] R. H. Netzer, "Optimal tracing and replay for debugging shared-memory parallel programs," *Proc. 3rd ACM/ONR Workshop on Parallel and Distributed Debugging,* 1993, pp. 1–11.

[34] N. Neves, M. Castro, P. Guedes, "A checkpoint protocol for an entry consistent shared memory system," *Proc. 13th ACM Symp. on Principles of Distributed Computing,* Aug. 1994.

[35] A. Nowatzyk *et al.*, "The S3.mp architecture: a local area multiprocessor," *Proc. 5th ACM Symp. on Parallel Algorithms and Architectures,* July 1993, pp. 140–141.

[36] G. G. Richard III and M. Singhal, "Using logging and asynchronous checkpointing to implement recoverable distributed shared memory," *Proc. 12th Symp. on Reliable Distributed Systems,* 1993, pp 58–67.

[37] R. E. Strom and S. Yemeni, "Optimistic recovery in distributed systems," *ACM Trans. on Computer Systems,* Vol. 3, No. 3, Aug. 1985, pp. 204–226.

[38] C. B. Stunkel *et al.*, "The SP1 high-performance switch," *Proc. Scalable High-Performance Computing Conf.,* May 1994, pp. 150–157.

[39] C. B. Stunkel, B. Janssens, and W. K. Fuchs, "Address tracing of parallel systems via TRAPEDS," *Microprocessors and Microsystems,* Vol. 16, No. 5, 1992, pp. 249–261.

[40] G. Suri, B. Janssens, and W. K. Fuchs, "Reduced Overhead Logging for Rollback Recovery in Distributed Shared Memory," *Proc. 25th Int. Symp. on Fault-Tolerant Computing,* June 1995.

[41] Y-M. Wang and W. K. Fuchs, "Optimistic message logging for independent checkpointing in message-passing systems," *Proc. 11th Symp. on Reliable Distributed Systems,* 1992, pp. 147–154.

[42] Y-M. Wang and W. K. Fuchs, "Lazy checkpoint coordination for bounding rollback propagation," *Proc. 12th Symp. on Reliable Distributed Systems,* 1993, pp. 78–85.

[43] K.-L. Wu, W. K. Fuchs, and J. H. Patel, "Error recovery in shared memory multiprocessors using private caches," *IEEE Trans. on Parallel and Distributed Systems,* Vol. 1, No. 2, April 1990, pp. 231–240.

[44] K.-L. Wu and W. K. Fuchs, "Recoverable distributed shared virtual memory," *IEEE Trans. on Computers,* Vol. 39, No. 4, Apr. 1990, pp. 460–469.